```
//pq.cpp
// Kevin Ly & James Le

#ifndef pq_cpp
#define pq_cpp
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

/*===================================================
MinPriorityQueue()             // default constructor
Precondition: None
Postcondition: An empty priority queue
===================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue()
{
        //capacity = 100;
        heapSize = 0 ;
        A = new KeyType* [capacity];
}

/*======================================================================
MinPriorityQueue(int n)        // construct an empty MPQ with capacity n
Precondition: Must be given a capacity size (n)
Postcondition: An empty priority queue with capacity of n
======================================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(int n)
{
        capacity = n;
        heapSize = 0;
        A = new KeyType* [n];
}

/*=========================================================================
MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);  // copy constructor
Precondition: Must be given a priority queue pq
Postcondition: Traverses the priority queue and makes a copy of its values
        to transfer to another priority queue
========================================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(const MinPriorityQueue<KeyType>& pq)
{
        heapSize = pq.heapSize;
        capacity = pq.capacity;
        A = new KeyType*[capacity];

        for (int i=0; i < heapSize; i++){
                A[i] = pq[i];
        }

        // buildHeap()
        heapSize = capacity;
        for (int i = (capacity/2); i >= 0; i--)
                heapify(i);
}

/*=================================================================
KeyType* minimum() const              // return the minimum element
Precondition: A non-empty min-heap A
```

```
Postcondition: Returns the minimum value in min-heap A
=================================================================*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::minimum() const
{
        if (empty())
                throw EmptyError();
        return A[0];
}

/*==========================================================================
KeyType* extractMin()            // delete the minimum element and return it
Precondition: A non-empty min-heap A
Postcondition: Deletes the minimum value in min-heap A and returns it
==========================================================================*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::extractMin()
{
        if (empty())
                throw EmptyError();
        KeyType* min = (A[0]);
        A[0] = A[heapSize-1];
        heapSize--;
        heapify(0);
        return min;
}

/*============================================================================
void decreaseKey(int index, KeyType* key)  // decrease the value of an element
Precondition: A min-heap A where new key is always smaller than current key
Postcondition: The value of element index’s key has the new value key
============================================================================*/
template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey(int index, KeyType* key)
{
        if (*(A[index]) < *key)
                throw KeyError();
        A[index] = key;
        while ((index > 0) && (*(A[index]) < *(A[parent(index)]))) {
                swap(index, parent(index));
                index = parent(index);
        }
}

/*==============================================================================
void insert(KeyType* key)                  // insert a new element
Precondition: Input is the key of the new element to be inserted into min-heap A
Postcondition: Key of the new node is in correct value and the heap maintains
        its min-heap property
==============================================================================*/
template <class KeyType>
void MinPriorityQueue<KeyType>::insert(KeyType* key)
{
        if (heapSize == capacity)
                throw FullError();

        if(heapSize ==0){
                A[heapSize] = key;
                heapSize++;
        }
        else{
        A[heapSize] = key;
        decreaseKey(heapSize, key);
        heapSize++;
        }
}


/*=========================================================================
bool empty() const                       // return whether the MPQ is empty
Precondition: None
Postcondition: Returns true if the priority queue is empty, false otherwise
=========================================================================*/
template <class KeyType>
bool MinPriorityQueue<KeyType>::empty() const
{
        if (heapSize == 0)
                return 1;
        else
                return 0;
}


/*=====================================================
int length() const         // return the number of keys
Precondition: None
Postcondition: Returns the length of the priority queue
=====================================================*/
template <class KeyType>
int MinPriorityQueue<KeyType>::length() const
{
        return heapSize;
}


/*================================================================================
std::string toString() const          // return a string representation of the MPQ
Precondition: A priority queue to be converted to a string
Postcondition: Traverses the array and uses << to output each element of the array
================================================================================*/
template <class KeyType>
std::string MinPriorityQueue<KeyType>::toString() const
{
        stringstream result; //sets variable to be returned
        int x = 0;
        result << "[";
        int size = heapSize;

        while(x < size) // inserts values into "result" while traversing list
        {
                result << *(A[x]);
                x++;
                if(x != size)
                        result << ",";
        }
        result << "]";
        return result.str();
}

/*================================================================================
std::string toString() const          // return a string representation of the MPQ
Precondition: A priority queue to be converted to a string
Postcondition: Traverses the array and uses << to output each element of the array
================================================================================*/
template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq)
```

```
{
        stream << pq.toString();
  return stream;
}


#endif
```

```
// test_pq.cpp
// Kevin Ly & James Le

#include <iostream>
#include <cassert>
#include "pq.h"

using namespace std;


void test_pq()
{
        int* empty[9];
        int A[] = {3,2,5,4,1,7,8,6,9};

        for(int i=0; i<9; i++){
                empty[i] = &A[i];
        }
        MinHeap<int> heapA(empty, 9);

        int x = 5;
        int y = 2;
        int z = 4;
        int a = 1;
        int b = 3;
        MinPriorityQueue<int> pq(10);

        pq.insert(&x);
        pq.insert(&y);
        pq.insert(&z);

        cout << "length is: " << pq.length() << endl;
        cout <<  "min is: "<< *pq.minimum() << endl;
        cout << pq.toString() << endl;
        cout << "decrease 5 to 1 " << endl;
         pq.decreaseKey(1, &a);
        cout << pq.toString() << endl;
        cout << "decrease 4 to 3 " << endl;
         pq.decreaseKey(2, &b);
        cout << pq.toString() << endl;

}






int main()
{
        test_pq();


        return 0;
}
```

```
// node.h
// Kevin Ly & James Le

#include <string>
#include <fstream>
#include <iostream>
#include <sstream>
#include <vector>
#include "pq.h"
using namespace std;

#ifndef NODE_H
#define NODE_H

//template <class KeyType>
class MinHeapNode
{
private:


public:


  char data; // One of the input characters
  string code;
  unsigned freq; // Frequency of the character
  MinHeapNode * left;
  MinHeapNode * right;
  MinHeapNode * parent;
  //,  right,  parent; // Left and right child of this node

//================================================
//Default Node Constructor
//================================================
  MinHeapNode()
  {
    data = ’\0’;
    freq = 0;
    left = NULL;
    right = NULL;
    code = "";
  }


/*================================================
MinHeapNode* newNode(char character, unsigned frequency)
Precondition: Requires a char character and a int frequency
Postcondition: Allocates a newNode with data=character and freq=frequency
================================================*/
MinHeapNode* newNode(char character, unsigned frequency) // construct node
  {
    MinHeapNode* n;

    n = new MinHeapNode();
    n->data = character;
    n->freq = frequency;
    n->left = NULL;
    n->right = NULL;
    n->code = "";
    return n;

  }
/*================================================
std::string toString() const
Converts Node into string output: [data: frequency]
================================================*/
std::string toString() const // return string representation
{
    stringstream result;
                result << "[" << data << ":" << freq << "]";
    return result.str();
}

/*================================================
bool operator < (MinHeapNode n)
Overwrites < to compare frequencies
================================================*/
bool operator < (MinHeapNode n) {// override function
    return (freq < n.freq);
  }
};

//template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinHeapNode& n) // stream operator
{
  stream << n.toString();
  return stream;
}


//#include  "huffman.cpp"

#endif
```

```
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string>
#include <vector>
#include "node.h"
#include "pq.h"
// Kevin Ly & James Le


using namespace std;

/*==================================================
void makecode(MinHeapNode* Root, string Arr[], int top)
Precondition: Requires a node (which is the root of the tree),
                                                        an array of strings, and an integer
Postcondition: Traverses the tree based on presence of
                                                        left child and right child. If left, 0
 is added to array.
                                                        if right, 1 is added to arra. Recursiv
ely calls itself
                                                        until it reaches a leaf. That node’s c
ode is equal to
                                                        the array.
================================================*/
void makecode(MinHeapNode* Root, string Arr[], int top){

string tempcode;
        if (Root->left != NULL){
                Arr[top] = "0";
                makecode(Root->left, Arr, top+1);


        }
        if (Root->right != NULL){
                Arr[top] = "1";
                makecode(Root->right, Arr, top+1);

        }
        if((Root->right == NULL) and (Root->left == NULL)){
                for(int i=0; i < top; i++){
                        tempcode += Arr[i];
                }
                Root->code = tempcode;
        }
}

/*==================================================
MinHeapNode* buildTree(MinPriorityQueue<MinHeapNode> &pq)
Precondition: Requires a priority queue
Postcondition: Extracts the two minimum nodes based on
                                                        their frequency and adds them together
 to make
                                                        a new node whose children are the node
s extracted.
                                                        The new node is then added to the pq.
                                                        Continues until only root is left in p
q.
                                                        Returns the root of the tree.
================================================*/
MinHeapNode* buildTree(MinPriorityQueue<MinHeapNode> &pq)
{
```

```
//cout << "======Start of buildTree========" << endl;
MinHeapNode* leftnode;
MinHeapNode* rightnode;
MinHeapNode* topnode;

// Iterate while size of priority queue doesn’t become 1
while (pq.length() != 1)
{

        MinHeapNode temp;

        leftnode = pq.extractMin();
        rightnode = pq.extractMin();
        topnode = temp.newNode(’\0’, (leftnode->freq + rightnode->freq));

        topnode->left = leftnode;
        topnode->right = rightnode;
  pq.insert(topnode);


}
                return pq.extractMin();
}

/*================================================
std::vector<MinHeapNode> compress(char *in, char *out)
Precondition: Requires a file to be compressed and a file to
                                                        be compressed to.
Postcondition: Produces a file made up of a string of 1’s
                                                        and 0’s. Traverses the in file and cal
culates
                                                        the frequency of each character.The no
de,
                                                        with character and frequency, is then 
added
                                                        to a vector. Each node is then added t
o a priority
                                                        queue. A tree is built based on the pq
. Calls
                                                        makcode function to assign strings of 
1’s and 0’s
                                                        to represent each character. These str
ings are then
                                                        added to the out file. Header of out f
ile gives each
                                                        character and their codes.
================================================*/
std::vector<MinHeapNode> compress(char *in, char *out)
{
        MinHeapNode* RootNode;
        std::vector<MinHeapNode> vect;
        bool contains = false;

        MinHeapNode node;
                ifstream in_file ( in );//  argv[2] is a filename to open

                if ( !in_file.is_open() )//  see if file opened
                        cout<<"Could not open file\n";
                else {
                        char x;
                        while ( in_file.get ( x ) ){ //returns false if end of file is reached
                                node.data = x;
                                for(int i = 0; i < vect.size(); i++){
                                        contains = false;
                                        if(x == vect[i].data){
                                                node.freq++;
                                                vect[i].freq++;
                                                contains = true;
                                        }
                                        if(contains == true)
                                                break;
                                }

                                if((vect.size() == 0) or (contains == false)){
                                        node.freq = 1;
                                        vect.push_back(node);

                                }
                                }
                                in_file.close();
                        }

//cout << "======Priority Queue======" << endl;
MinPriorityQueue<MinHeapNode> pq;
for (int k = 0; k < vect.size(); k++){
        pq.insert(&vect[k]);
}

RootNode = buildTree(pq);

MinHeapNode tempNode;
tempNode = *RootNode;
string Arr[100000];
int top = 0;

makecode(RootNode, Arr, 0);
//=======Put in out_file========
ofstream out_file ( out );
ifstream in_file2 ( in );
char x;

for(int j = 0; j < vect.size(); j++){
        if(x = vect[j].data)
                out_file << vect[j].data << vect[j].code <<".";
}
out_file << "*";


while(in_file2.get(x)){
        for(int j = 0; j < vect.size(); j++){
                if(x == vect[j].data)
                        out_file << vect[j].code;
        }
}

return vect;
}

/*==================================================
void decompress(char *in, char *out)
Precondition: Requires a file to be decompressed and a file to
                                                        decompressed to.
Postcondition: Reads the in file and makes nodes based on
                                                        the header (gives character and code).
 When it
                                                        encounters a '*', it inserts the follo
```

```
wing
                                                        1’s and 0’s into an array. The code fo
r each
                                                        character is then compared to the elem
ents
                                                        of the array. When a code corresponds
to a key
                                                        the index of the key is saved and the
following
                                                        elements are compared to a new (or cur
rent) character
                                                        code. Each time a code is read, it is
outputted
                                                        into the out file.
==================================================*/
void decompress(char *in, char *out)
{
MinHeapNode* RootNode;
std::vector<MinHeapNode> vect;
        bool go = true;
        bool hit = false;
        string temp;
        MinHeapNode node;
        string tempA[100000];
        int A_count = 0;


                ifstream in_file ( in );//  argv[2] is a filename to open

                if ( !in_file.is_open() )//  see if file opened
                        cout<<"Could not open file\n";
                else {
                        char x;
                                        while(in_file.get ( x )){
                                                if(x != ’*’){
                                                while(go == true){
                                                if(x != ’1’ and x != ’0’ and x != ’.’){
                                                        temp = ’\0’;
                                                        node.data = x;
                                                        break;
                                                }
                                                else if (x == ’1’  or x==’0’){
                                                        temp += x;
                                                        break;
                                                }
                                                else if (x==’.’){
                                                        node.code = temp;
                                                        vect.push_back(node);
                                                        go = false;
                                                }
                                                }
                                                go = true;
                                        }

                                                if(x == ’*’){
                                                        hit = true;
                                                }
                                                if ((hit == true) and (x == ’*’)){

                                                }
                                                else if((hit == true) and ((x == ’1’) or (x ==
 ’0’ ))){

                                                        tempA[A_count] = x;
                                                        A_count++;
                                                }
                                        }


string tempcode;
string read = "";
ofstream out_file ( out );
int counting = 0;
int maxsize=0;
while(counting != A_count){
                for (int i = 0; i < vect.size(); i++){
                        tempcode = (vect[i].code);
                        string code = "";

                        for (int t=0; t < tempcode.size()-1; t++){
                                code += tempcode[t+1];
                                if (code.size() > maxsize)
                                        maxsize = code.size();
                        }

                        while((vect[i].code != read) and (counting < A_count)){
                                read += tempA[counting];
                                if( code == read){
                                        out_file << vect[i].data;
                                        break;
                                }
                                if(read.size() >= maxsize)
                                        break;

                                counting++;
                        }

                        if(counting == A_count){
                                break;
                        }
                        else if(code == read){
                                counting++;
                                read = "";
                                break;
                        }
                        else if( (read.size() >= maxsize)){
                                counting = counting - read.size() + 1;
                                read = "";
                        }


                }

}

        }

                                in_file.close();
}

//=====End of Decompressor======================
int main(int argc, char *argv[])
{
        char *temp;
        std::vector<MinHeapNode> vect;
        if (argc == 4){
                temp = argv[1];
                char blank = temp[1];

                if (blank == ’c’)
                        vect = compress(argv[2], argv[3]);

                else if (blank == ’d’){
                        decompress(argv[2], argv[3]);
                }

                else{
                        cout << "Error: verify compression (-c) or decompression (-d)" << endl
;
                }
        }

}


 //end of main
```